# Choreographer Pre-Testing, Code Analysis, and Operational Testing

David J. Fritz, Christopher Harrison, C.W. Perr, and Steven Hurd
Sandia National Laboratories, Livermore, CA

July 14, 2014

## 1 Executive Summary

Choreographer is a “moving target defense system”, designed to protect against attacks aimed at IP addresses without corresponding domain name system (DNS) lookups. It coordinates actions between a DNS server and a Network Address Translation (NAT) device to regularly change which publicly available IP addresses' traffic will be routed to the protected device versus routed to a honeypot. More details about how Choreographer operates can be found in Section 2: Introducing Choreographer. Operational considerations for the successful deployment of Choreographer can be found in Section 3. The Testing & Evaluation (T&E) for Choreographer involved 3 phases: Pre-testing, Code Analysis, and Operational Testing.

Pre-testing, described in Section 4, involved installing and configuring an instance of Choreographer and verifying it would operate as expected for a simple use case. Our findings were that it was simple and straightforward to prepare a system for a Choreographer installation as well as configure Choreographer to work in a representative environment. Code Analysis, described in Section 5, consisted of running a static code analyzer (HP Fortify) and conducting dynamic analysis tests using the Valgrind instrumentation framework. Choreographer performed well, such that only a few errors that might possibly be problematic in a given operating situation were identified.

Operational Testing, described in Section 6, involved operating Choreographer in a representative environment created through Emulytics $^{TM}$. Depending upon the amount of server resources dedicated to Choreographer vis-à-vis the amount of client traffic handled, Choreographer had varying degrees of operational success. In an environment with a poorly resourced Choreographer server and as few as 50-100 clients, Choreographer failed to properly route traffic over half the time. Yet, with a well-resourced server, Choreographer handled over 1000 clients without misrouting. Choreographer demonstrated sensitivity to low-latency connections as well as high volumes of traffic. In addition, depending upon the frequency of new connection requests and the size of the address range that Choreographer has to work with, it is possible for all benefits of Choreographer to be ameliorated by its need to allow DNS servers rather than the end client to make DNS requests.

Conclusions and Recommendations, listed in Section 7, address the need to understand the specific use case where Choreographer would be deployed to assess whether there would be problems resulting from the operational considerations described in Section 3 or performance concerns from the results of Operational Testing in Section 6. Deployed in an appropriate architecture with sufficiently light traffic volumes and a well-provisioned server, it is quite likely that Choreographer would perform satisfactorily. Thus, we recommend further detailed testing, to potentially include Red Team testing, at such time a specific use case is identified.

## 2 Introducing Choreographer

Cyber attackers often have low success rates in their attacks: most systems are patched or have users that do not fall for online deception. Accordingly, these attackers make up for this in volume. Using large botnets, attackers can launch SQL injection or phishing attacks against thousands of organizations to scour for a single vulnerable system or user. In doing so, these attacker optimize for speed and throughput, cutting corners in their network-level activity.

Choreographer combats these botnet-based attacks by using a moving targets approach. The Choreographer system directs a domain name system (DNS) server and a network address translation (NAT) device to cooperate in randomizing the public facing addresses of protected servers. When a visitor requests the IP address of a particular Web server, Choreographer detects the request and authorizes the NAT device to allow access to the server through a particular IP address. When the user accesses the server, the NAT device will grant access. However, if an attacker accesses the server without using the DNS, the NAT device will instead direct the attacker to a "honeypot" system. The honeypot is a carefully monitored system that mimics the intended target, with purported vulnerabilities for the attacker to exploit. By analyzing the attacker's behavior on the honeypot, the organization can learn more about their cyber opponents. Further, if an alarm is tripped on a legitimate system, a connection can be seamlessly migrated to the honeypot without alerting the attacker.

Choreographer can also be used for internal systems, rapidly detecting outbreaks of malware or compromised systems being directed by an attacker. With additional settings, choreographer could additionally be used for Insider Threat detection.

### 2.1 Modes of Operation

Choreographer has two main modes of operation: a centralized architecture and a distributed architecture. In the centralized mode, a single process does everything: it watches for incoming DNS packets, it determines the IP rotations needed, it updates the DNS server and it updates the NAT tables. While straightforward, the centralized architecture does not allow more than one DNS server to be commanded and it can only handle one NAT device.

The distributed mode eliminates these requirements. In a distributed setting, a single instance of Choreographer still monitors traffic and makes IP rotation decisions, but it contacts agents to update the DNS and NAT infrastructure. This supports multiple authoritative DNS servers and multiple NAT devices. To perform this coordination, the system uses TCP sockets and authenticates the connection and each command using one-way hashes keyed with a preshared secret. This allows the agents to have confidence that they are interacting with the Choreographer and not an attacker.

## 3 Operational Considerations

Like virtually any technology, Choreographer has a range of use cases where Choreographer will work well and other use cases where Choreographer would not work well. This section describes operational considerations for Choreographer, so a potential adopter may determine whether Choreographer is appropriate for their specific use case.

Ideally, Choreographer would forward all legitimate traffic to its intended destination and direct all malicious traffic to the "honeypot". However, in a typical operational environment, it's reasonable to assume that there is some risk, however slight, of misdirection. Especially with respect to the misdirection of legitimate traffic, we recommend that an organization deploying Choreographer should identify the percentage of misdirection that is acceptable.

The operational considerations are a function of how Choreographer is deployed and configured. While there are an infinite number of permutations regarding deployment and configuration details, this section focuses on a single main implementation: Providing services to users on the internet.

### 3.1 DNS Considerations

In the case of providing services to users on the internet, there are a few important operational considerations with respect to domain name services. First, if there are DNS servers not being controlled by Choreographer in the DNS hierarchy above the DNS server Choreographer is controlling, that insist on being authoritative for the domain, Choreographer will almost certain not work properly.

In addition, the client side of the DNS hierarchy can cause operational problems for Choreographer. Many of the DNS requests that Choreographer uses to identify legitimate connections come from DNS servers rather than directly from the system initiating the connection request. To compensate for these indirect requests, Choreographer injects an 'open' firewall rule that will allow any host from any IP address to connect to the protected server. This open firewall rule will be in place until expiration of the Time to Live (TTL) value that is configured for the DNS zone. All connections during this window of time will result in an IP address-specific firewall rule being created. The TTL value is used both as the length of time Choreographer will allow the open firewall rule to be active as well as the value set for TTL included in the DNS reply to the DNS request. This TTL value is at the root of problems that can exist on the client side of the DNS hierarchy.

If a caching DNS server (not under the control of Choreographer) receives a second connection request during Choreographer's Time-to-Live (TTL) window, that second connection will be provided the same IP address provided to the first connection. In this case, the second client's attempt to connect using that original destination IP address will be routed to the honeypot if the TTL expires before this connection is established. However, given the fairly small time window where this is likely to occur, this specific scenario is

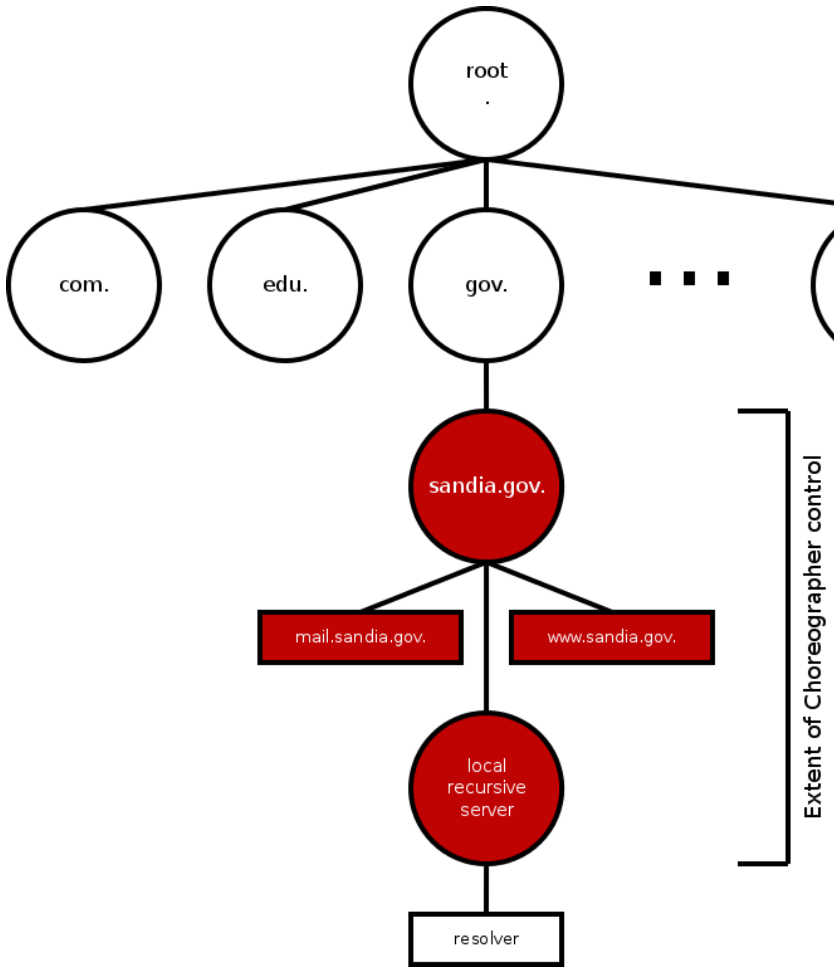

Figure 1: DNS hierarchy indicating the parts of DNS that Choreographer is likely to control.

deemed unlikely. Example, roughly speaking, if the TTL value was 5 seconds and typical round-trip propagation delays were on the order of 200 milliseconds, implying the one way delay is on the order of 100 milliseconds, the time where this would occur is for connection requests occurring between the 4.9 and 5.0 second marks, as connections occurring before and after this window would not be re-routed, because connections occurring before would be properly routed and connections occurring after would initiate a new DNS request to the Choreographer server, resulting in a new destination IP address provided as the DNS reply.

This artifact is also problematic in the scenario where a malicious connection takes place, without a valid DNS request, and is routed to the protected server incorrectly. Essentially, for every open firewall rule created, Choreographer removes its protection on that particular IP for the length of the DNS TTL. The problem may be amplified when many legitimate connections are made, and is discussed in the Access Window Aliasing section.

However, this scenario assumes that the caching DNS server was honoring the TTL value supplied by Choreographer. DNS servers are not strictly required to honor the TTL they are provided. If a longer TTL was substituted, any connections between the expiration of the TTL supplied by Choreographer and the TTL used by the caching DNS server would be routed to the honeypot, as the caching DNS server would be providing an expired address to queries that occur during that time. The testing team does not have definitive information on the prevalence of caching DNS servers that do not honor TTL values they receive from other DNS servers (such as Choreographer).

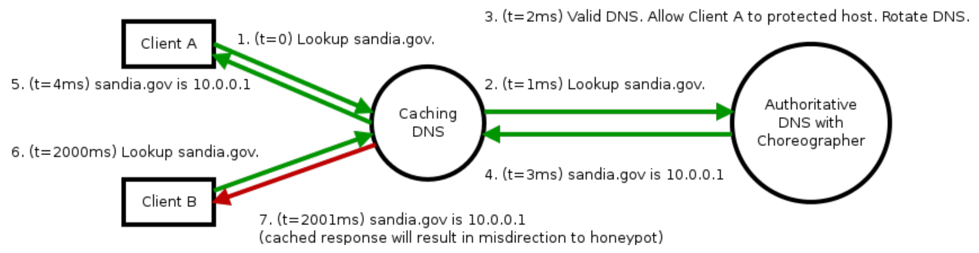
Client A
1. (t=0) Lookup sandia.gov.
5. (t=4ms) sandia.gov is 10.0.0.1
Caching DNS
2. (t=1ms) Lookup sandia.gov.
3. (t=2ms) Valid DNS. Allow Client A to protected host. Rotate DNS.
Authoritative DNS with Choreographer
4. (t=3ms) sandia.gov is 10.0.0.1
6. (t=2000ms) Lookup sandia.gov.
Client B
7. (t=2001ms) sandia.gov is 10.0.0.1
(cached response will result in misdirection to honeypot)

Figure 2: Two legitimate clients making DNS queries to a caching DNS server not under the control of Choreographer. If the second query arrives within the TTL of the first record, the second client will be redirected to the honeypot incorrectly.

Additionally, most operating systems use some form of host-level nameserver caching. These host-level tools often disregard DNS TTL entries and instead use TTL values configured at the host. For example, the nscd caching service, available on most GNU/Linux distributions, explicitly does not honor DNS TTL records and has a default TTL of 3600 seconds (1 hour) for any DNS reply.

Will these caching DNS issues impact a specific use case? The most appropriate answer is "it depends". Under the assumption that caching DNS servers respect Choreographer's TTL, redirection to the "honeypot" is largely limited to edge cases for connections occuring right about the time the TTL expires. If it is rare that two clients anywhere on the internet, much less using the same caching DNS server, attempt to connect during a single instance of the TTL, this almost certainly won't be a problem. In other use cases, detailed statistical analyses of server logs correlated with DNS server logs would be required to yield the answer.

## 3.2 DNSSEC

The United States Federal Government, with direction from the Department of Homeland Security (DHS), The National Institute of Standards and Technology (NIST), and the Office of Management and Budget (OMB) have mandated that all federal agencies switch to using DNSSEC. Choreographer, by design, frequently updates zones, and while under DNSSEC this does not generate data that must be sent upstream, and therefore can maintain the chain of trust, records must be resigned at every zone change. Furthermore, the NSEC3 extension to DNSSEC, which provides ”denial of existence” service, may have to traverse the served zones to recompute and resign the NSEC3 record.

The cost of zone signing factors into the maximum rate at which Choreographer can update zones. For example, we timed resigning a simple zone 1000 times in succession. The zone was signed with NSEC3 capable, 2048-bit, SHA256 keys (the required algorithm and minimum length as defined by current DNSSEC standards). The machine used was a 24-core Xeon 2630 machine, with 128GB of RAM. The machine was booted with a custom init script to ensure only sshd and bash were running (no other non-kernel processes were running). The time required to complete 1000 zone signings was 40.3 seconds, or 40.3 milliseconds per signing. This cost should be noted when deploying Choreographer.

## 3.3 Access Window Aliasing Considerations

As noted in an earlier section regarding DNS considerations, DNS requests often come from a caching DNS server rather than the actual end client system. Choreographer injects an ’open’ firewall rule that allows any host access until TTL expires. This functionality creates a window of opportunity for any client to attempt to connect using that IP address, even if the DNS request was made by another client. This results in allowing service to a potential attacker.

To mitigate this phenomena, Choreographer rotates the IP address associated with the DNS reply, which adds the difficulty of having to guess the IP address that answers the DNS request. Each new connection request made to DNS will result in a new IP address being provided with the DNS reply, except if the minimal IP rotation parameter is used.

The likelihood of a specific window of vulnerability for an adversary to exploit this is a function of (1) the number of legitimate connection requests, (2) the DNS TTL, (3) the frequency that Choreographer rotates addresses (configurable) and (4) the size of the address space used for rotating IP addresses.

As an example, if (1) the number of legitimate connection requests was 600/minute (or 10/second), (2) each open firewall rule was open for 5 seconds, (3) Choreographer may rotate the address after 1 second, and (4) the available address space included 200 addresses, the following ratio (2):(3) would determine how many IP addresses are available for connection without requiring a DNS lookup at a given time. In this case, that would result in 5 addresses of the 200 addresses being open at any time, and approximately 200 seconds being the time required to successfully establish connections to all 200 IP addresses.

The likelihood of these occurrences is reduced when the size of the address space is increased.

## 4 Pre-Testing

Testing began by first reading the documentation and seeing that Choreographer has two main modes of operation. A centralized architecture, and a distributed architecture. The centralized architecture was chosen to test first.

> ”In the centralized mode, a single process does everything: it watches for incoming DNS packets, it determines the IP rotations needed, it updates the DNS server and it updates the NAT tables. While straightforward, the centralized architecture does not allow more than one DNS server to be commanded and it can only handle one NAT device.”

With this information we started to construct the test bed which would be handled virtually. A base Debian 7 64-bit image was created as a base for the choreographer and other machines.

The following libraries were installed on the choreographer machine once it was started.

- libpcap-dev
- libbotan1.8-dev

Some specific packages were also included on the other machines to enable the test environment or other aspects of the test.

The basic network topology was set up to mirror the provided documentation using virtual machines. The documentation provided the topology in Figure 3

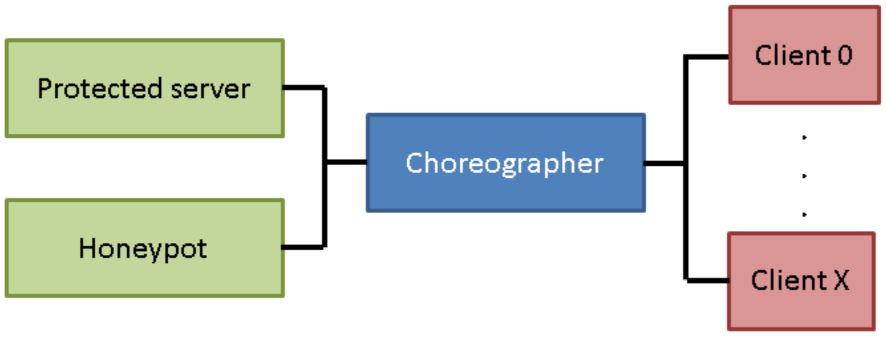

Figure 3: Choreographer Emulytics$^{TM}$ network topology. Choreographer runs on the center NAT along with BIND.

The following addresses and address spaces were used for the test, with virtual machines created for the webserver, the honeypot, choreographer, and client nodes.

facebook.com was used as the web server with IP address for facebook.com as 69.63.176.2 The honeypot was given 69.63.176.3, the DNS/DHCP/Choreographer system was given 69.63.176.1/8.8.9.1, with the Choreographer back-end interface (eth1) given 69.63.176.1, the Choreographer front-end interface (eth0) given 8.8.9.1, and the test VMs given a range from 8.8.0.0/16 (serving 8.8.8.0/24).

To first test the set up, without choreographer running, the clients were tested to see if they could connect to facebook by using 'wget' and 'ping' commands. Once it was established that the network connections were working as desired choreographer was installed to be tested.

As recommended in the documentation the file, choreographer.conf, was edited before compilation in order to fit the test that we were creating. We attempted to follow the example files provided as closely as possible.

**REDACTION NOTICE: The configuration file and any other Choreographer code is considered proprietary information and has been removed from this unlimited release report.**

At this point Choreographer performed as expected with limited troubleshooting and configuration with assistance from the the developers. Initial deployment of Choreographer was non-trivial, but given the stage of development that Choreographer was tested at the problems encountered were not atypical.

## 5 Code Analysis

Source Code Analysis was conducted via HP's Fortify and Valgrind.

Fortify is described by HP in the following:

> HP Fortify Static Code Analyzer helps verify that your software is trustworthy, reduce costs, increase productivity and implement secure coding best practices. Static Code Analyzer scans source code, identifies root causes of software security vulnerabilities and correlates and prioritizes results—giving you line–of–code guidance for closing gaps in your security. To verify that the most serious issues are addressed first, it correlates and prioritizes results to deliver an accurate, risk–ranked list of issues.

More information on Fortify can be found at: http://www8.hp.com/us/en/software-solutions/software-security/

The main rule sets used by Fortify to analyze C++ source code are mainly developed using the most common vulnerabilities and up to date rule sets.

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android (2.3.x and later), X86/Android (4.0 and later), X86/Darwin and AMD64/ Darwin (Mac OS X 10.7, with limited support for 10.8).

More information on Valgrind can be found at: Valgrind (http://valgrind.org/)

### 5.1 Fortify

The audited Fortify Report is provided to the technology providers in a separate standalone report.

4 critical errors were found which were determined to be of low risk given how Choreographer is deployed. The common issue in all of the critical errors was the possibility of a command injection. An attacker would only be able to execute this command if they already had access to a system, in which case there would be much greater issues to worry about.

43 high level issues were found, of which 22 were found to be non-issues. 2 were found to be possible reliability issues as 2 functions were identified which failed to release system

resources. The remaining 19 we considered bad practice errors as they calls which allocated memory but failed to free it. Solutions to all of these issues are recommended in the body of the Fortify report.

42 low level issues were found, of which 9 were found to be non-issues. 1 issue was decided to be 'suspicious', but will need to be considered by the developers. The remaining issues were all considered 'bad practice' as they mostly dealt with calls to possibly deprecated functions, failure to check against null values in the case of malloc errors, and variables which were never read. Recommendations were made to correct most of these issues to help increase the overall reliability of the software.

## 5.2 Valgrind

Dynamic code analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. In this subsection we describe the dynamic analysis of Choreographer code using Valgrind. We first give an overview of the types of issues that Valgrind attempts to detect and then list the results of our tests. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior.

The Valgrind framework is a powerful tool to debug your applications, and especially for memory allocation related bugs. You can get a detailed explanation of all VALGRIND tools in the main site of the project (Valgrind (http://valgrind.org/)) and the Valgrind User manual (http://valgrind.org/docs/manual/manual.html). Valgrind can be used to analyze a number of language families (including the C and Java families found in Choreographer).

### 5.2.1 Memory leaks

Blocks of statically allocated memory, those available in the `stack` of the process, will be available as long as the program runs, but only in the specific execution context of each moment. For example, variables declared at the beginning of a given function will only be available as long as the execution stays inside that specific function (or in a lower context). You can view as if those blocks of memory in the `stack` are automatically de-allocated for you during the execution of the program when moving to upper contexts.

On the other hand, you can dynamically allocate a block of memory in the `heap` inside a given function, and return the pointer to that block of memory (address of the block) to the upper context, making it available outside of the context where it was originally allocated. The GNU C LIBRARY provides several methods to manage this memory in heap, but the most common ones are `malloc` and its variants[1] (in `malloc.h`), which provide an unconstrained way of allocation in the heap:

```
void *malloc(size_t size);
```

[1]Or any other method which uses `malloc()` internally, like `strdup()`

```
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

In C, the programmer is in charge of explicitly de-allocating that block of memory when it no longer will be used. Other programming languages implement a mechanism to handle this de-allocation requirement, usually called a `garbage collector`. Thus, **a memory leak occurs when a block of dynamically allocated memory**(using `malloc()` for example) **is never freed/deallocated explicitly** (with `free()`).

### 5.2.2 Valgrind Findings

The reports generated by Valgrind are available separately from this report. The reports generated are quite lengthy, difficult to view and comprehend, and repeat similar finding many times over. To summarize Valgrind's main findings, there were 651 memory errors from 651 different contexts. 6 of those errors were considered to be suppressed. That said, it is the opinion of the testers that the main memory leaks were not the fault of the developers, but rather resulting from the use of libpcap and libstdc++. However, being integral to Choreographers functionality, it may be worthwhile to examine Valgrind's output to identify memory leaks which affect scalability.

## 6 Operational Testing

### 6.1 Operational Testing Environment Information

Along with code analysis, a detailed in-situ evaluation of Choreographer was performed using minimega, a large scale Emulytics$^{TM}$ platform designed to provide a virtualization based emulation of arbitrary network topologies. Minimega can launch up to 100 thousand endpoints consisting of Linux, Windows, and Android devices, and can provide representative traffic generation to and from all endpoints. Additionally, minimega can facilitate attaching real hardware to an emulated experiment (such as a transparent mail filter, router, etc.) at any point within the network topology.

#### 6.1.1 Network Topology

**REDACTION NOTICE: The configuration file and any other Choreographer code is considered proprietary information and has been removed from this unlimited release report. This included the Emulytics $^{TM}$ platform used in testing Choreographer**

Choreographer is designed to run on a Network Address Translation (NAT) device, which behaves like a router but masks the IP address of hosts behind the NAT. The smallest representative network topology designed to work with Choreographer consists of the Choreographer device situated as a NAT in front of at least two servers - a protected server and a honeypot. On the other side of the Choreographer device are a number of clients making requests to access the protected server, whose IP address is masked by the Choreographer NAT. Figure (redacted) shows our experimental topology.

Invalid requests made to the protected server will be automatically rerouted to the honeypot based on the policies of Choreographer. Valid requests, those made by first making a DNS lookup to the host in question, are routed to the protected server. Choreographer must then be able to control an instance of a DNS server, in particular BIND. Choreographer supports a distributed model in which one or more BIND servers can be controlled remotely by Choreographer, allowing the decoupling of DNS service from the Choreographer NAT device.

To simplify the experiment, we chose to run BIND locally on the Choreographer NAT, and run Choreographer without external agents. This allows us to focus on evaluating the core functionality of Choreographer without introducing the extra complexity of its distributed model.

The Emulytics$^{TM}$ environment can create impractical network timing scenarios including sub-microsecond latency between hosts. To provide a more representative environment, we inject an artificial latency into the network based on a truncated normal distribution with u=20ms and sigma=5ms. We also ran all tests under a high latency model using u=200ms and sigma=5ms. The 20ms and 200ms means represent the ends of the most credible portion of the latency spectrum in practice.

Although the timing distribution used here supports injecting zero additional latency, a zero latency packet transmission cannot exist in our platform. If the distribution injects no latency, the packet will arrive at the destination host with a latency equal to that of the underlying physical network of the Emulytics$^{TM}$ platform (generally less than 1ms), which is not realistic.

### 6.1.2 Traffic Generation

To provide Choreographer with data to consume, we use custom traffic generators run on each client, and custom services run on both the protected server and honeypot. Once the response to the DNS query has been received and processed, we generate HTTP, HTTPS, SSH, and SMTP traffic from the clients to a single domain. The traffic generators perform legitimate DNS lookups in order to generate requests. To evaluate invalid traffic, the traffic generators can be configured to access the protected server directly without first making a DNS request.

The traffic generators randomly make HTTP, HTTPS, SSH, and SMTP transactions based on a normal distribution with u=1s and sigma=1s following the end of the prior transaction. It is possible for the normal distribution to result in negative values, which were again substituted with a 0 value.

### 6.1.3 Host Configuration

All clients run a custom Linux distribution with 1 CPU, 2GB of RAM, and a single 1Gb interface. The traffic generator software takes a configuration passed on the kernel command line at boot time.

All routers run the Vyatta router operating system, have 1 CPU, 2GB of RAM, and at least 2 1Gb interfaces. All routers use OSPF for dynamic routing.

The device runs Debian 7.4, with BIND installed and configured to serve a single record (facebook.com). The DNS record is configured with a TTL of 5 seconds. This value was chosen to alleviate window aliasing concerns (discussed later). It should be noted that a 5 second TTL is considered exteremely low, with Disaster Recovery (DR) systems generally implementing 30 second TTLs.

## 6.2 Operational Testing Methodology

We ran 9 configurations of scale and traffic mix with each of the latency models described above, resulting in 18 total tests. The configurations include 50, 100, and 1000 clients, as well as all valid, no valid, and some valid traffic for each scale. Each test was run for 5 minutes, allowing up to several hundred thousand transactions to take place. All traffic was logged at each client as well as at the server.

Each endpoint in the experiment is a 64-bit Intel-based VM with one CPU and 2GB of RAM. Each endpoint has a 1Gb link to a 10Gb experiment backplane. A second round of tests was performed with each endpoint given a 10Gb link with a 30Tb backplane. On the second round of tests, the Choreographer device was also given 16 CPUs and 16GB of RAM. The traffic generators are throttled to prevent saturating links.

While Choreographer supports a distributed mode which allows controlling one or more Choreographer installations remotely, we chose to focus operational testing efforts on the local mode. This choice was made because the local mode is simpler to configure, control, and monitor, and issues discovered in the local mode will generally also apply to the distributed mode.

In general tests were conducted to investigate a breadth of potential issues. Those issues raised here may require additional investigation to evaluate criteria for implementation and deployment, either through additional testing by Sandia, or by the software authors.

## 6.3 Operational Testing Results

A number of potential issues were discovered during the scaled tests. Some of these issues are artificially more dramatic than we expect to see in a real environment, such as a sensitivity to latency, but remain salient and are documented here. Additionally, the virtualization resources allocated to the Choreographer device (which is run as a Virtual Machine) are artificially limited to induce stress on the device from the clients. Details are given below, but it is important to note that limits of scale described below can be moved, but not eliminated, by providing more capable resources to Choreographer.

As noted above, a second round of tests was run to give the Choreographer significantly more resources under the same load. These tests completed with better results and are noted below.

### 6.3.1 Sensitivity to Latency

Choreographer requires a non-zero amount of time after noting a valid DNS request to inject a new firewall rule for the client making the request. If the client is able to make a request to the protected server after a valid DNS request before Choreographer can inject a rule for that client, the client request will be routed to the honeypot incorrectly instead.

In practice, this is far less of an issue than in our Emulytics$^{TM}$ environment, as it is capable of allowing clients to make requests with very little latency. We chose to test with two latency values: 20ms and 200ms. These represent the ends of the most credible portion of the latency spectrum in practice. After artificially injecting latency into the environment, as described above, we saw no more issues related to the unrealistically short latency times resulting from the Emulytics$^{TM}$ environment.

The amount of latency between requests is dependent on how quickly Choreographer can parse the DNS request and inject a firewall rule. This is in turn dependent on the CPU performance of the Choreographer device.

In both sets of tests (low and high resource availability), Choreographer demonstrated sensitivity to latency.

### 6.3.2 Access Window Aliasing

As discussed in the earlier section regarding operational considerations, to compensate for indirect DNS requests such as those made by a caching DNS server, Choreographer injects an 'open' firewall rule for all hosts for a period of time (the TTL of the DNS record) after the valid DNS request is made. This allows the client to make the following request to the protected server successfully, at which point Choreographer creates a rule unique to that client as well as any other client who make a connection before the TTL expires. To combat this, Choreographer rotates the IP address associated with a DNS record, adding the difficulty of having to guess the IP address given from the original DNS request.

DNS service on the internet typically has a TTL measured in minutes or hours, but can be as low as several seconds at the expense of additional load on authoritative DNS servers. To balance the operational considerations, at the suggestion of the technology providers, all testing used a DNS TTL of 5 seconds, which is several orders of magnitude shorter than typical TTL values found in DNS replies on the internet.

For a DNS record with a 5 second TTL, this means that Choreographer will inject an 'open' rule for a particular destination IP for no more than 5 seconds, allowing the requester time to make the connection to the protected server. Assuming all open rules stayed active for 5 seconds with no minimum time set for IP rotation, if Choreographer is protecting a host using a /24 IPv4 block, this means that once 256 * (60/5) = 3072 requests/minute are made (assuming they are made uniformly over that minute), Choreographer will have an 'open' rule for every IP in the /24 block to all clients. This in effect causes Choreographer to 'fail open'.

This effect can be minimized by lowering the TTL and raising the size of the masking IP space. It may be impractical to use anything larger than a /24 IPv4 block given the dramatic resource limits of IPv4. However, the typical allocation in IPv6 is a /64, which in the 5 second TTL example above, would require 221360928884514619392 requests/minute to create the same aliasing effect.

### 6.3.3 Sensitivity to Volume of Traffic

Like any network connected device, Choreographer is sensitive to Denial of Service attacks. Choreographer is unique however in how it responds to a heavy traffic load.

Choreographer uses libpcap to capture DNS traffic in order to make decisions on firewall rule injection. libpcap is known to be lossy at high traffic volume, which would impact the efficacy of Choreographer. In effect, if Choreographer misses a valid DNS request due to libpcap dropping buffered packets, then the client, which should be considered legitimate, will hit the default firewall rule and be routed to the honeypot.

Table 1: Ratio of incorrectly routed requests to total requests with a resource limited Choreographer.

| Number of clients | 20ms | 200ms |
|---|---|---|
| 50 | 11308/16890 (66%) | 0/7694 (0%) |
| 100 | 22696/23693 (96%) | 10693/15212 (70%) |
| 1000 | 194839/202514 (96%) | 105789/107733 (98%) |

Important Reminder: The initial set of results came from testing using a Choreographer server that was resource-constrained, having only 1 CPU and 2GB of RAM available.

Table 1 illustrates this phenomenon. All traffic generated in the data in Table 1 is normal, valid traffic. At 50 clients and a 200ms latency from client to server, no packets are lost in Choreographer, and all traffic is correctly routed to the protected server. However, when 100 clients are used, nearly 70% of valid traffic is incorrectly routed to the honeypot. This becomes far more dramatic with the number of clients.

In the second round of tests, the Choreographer device was given 16 CPUs and 16GB of RAM with 10GB interfaces. Under the same load, Choreographer lost no packets and no traffic was incorrectly routed to the honeypot. This is illustrated in Table 2.

Table 2: Ratio of incorrectly routed requests to total requests with more capable Choreographer hardware.

| Number of clients | 20ms | 200ms |
|---|---|---|
| 50 | 0/32119 (0%) | 0/7510 (0%) |
| 100 | 0/57961 (0%) | 0/14930 (0%) |
| 1000 | 0/159604 (0%) | 0/161753 (0%) |

To further illustrate the issues Choreographer has at scale, we re-ran one of the tests using larger numbers of clients, up to 7500. As the number of clients increased, the amount of traffic being incorrectly routed to the honeypot increased. With approximately 5000 clients, the Choreographer device began issuing nf_conntrack errors, indicating the connection tracking table used for stateful NAT connections in the kernel was overflowing. After increasing the size of the table dramatically, to one million entries, and lowering the timeouts on key nf_conntrack fields, the table stopped overflowing, but was still slowly increasing in size, and would still overflow in a matter of hours. For this test, we updated the nf_conntrack parameters with those listed in Table 3.

Table 3: nf_conntrack parameters used in scaled testing.

| Field | Value |
|---|---|
| net.netfilter.nf_conntrack_generic_timeout | 60 |
| net.netfilter.nf_conntrack_tcp_timeout_established | 60 |
| sys/module/nf_conntrack/parameters/hashsize | 250000 |
| net.netfilter.nf_conntrack_max | 1000000 |

## 6.4 Firewall Limits

Choreographer injects firewall rules into the NAT chain for every allowed client. The maximum number of rules possible in a single chain is limited by the available kernel memory and the size of each constructed rule, but in practice issues occur well before the kernel memory allocation is reached. For example, a ruleset of 25000 Choreographer injected rules will consume over 2GB of kernel memory. Each time a new rule is injected, the kernel copies the entire table, injects the new rule, loads the new table, and frees the old table. In the interim, the kernel requires over 4GB of memory to accommodate the rule injection.

Other limits, such as the size of the stateful connection tracking table, will almost certainly be reached before any firewall table size limits.

## 6.5 Operational Testing Conclusions

It should be noted that the exact amount of incorrectly routed traffic is dependent on the performance of the Choreographer device. In the first round of experiments, Choreographer was run with a single CPU within a Virtual Machine. When run again with more capable hardware and the same amount of traffic, the proportion of incorrectly routed traffic goes to zero. However, even with more capable hardware, the issue remains, albeit with different limits.

There exist other issues at scale beyond the ability of Choreographer to capture and process all incoming DNS traffic. Choreographer requires the use of a NAT, which maintains state for all connections. This can be extremely problematic at scale, as the kernel will drop packets from reaching the NAT chain of the firewall if the connection tracking table is full.

These issues are not fundamentally different from those many system administrators face. Often the solution to these issues include greater hardware resources and fanning load out to many servers. An administrator using Choreographer would need to take care to ensure that the Choreographer system had enough resources to correctly handle the expected load of traffic.

## 7 Conclusions and Recommendations

Choreographer performed as expected, and could be a valuable protection in certain environments but with some serious limiting factors. The code for Choreographer is relatively simple and straight-forward.

The issues which face Choreographer are significant, yet from our testing, we believe they are possible to overcome given an adequately provisioned system as well as an external system that mitigates DoS attacks or unexplained usage spikes. Some further testing might be useful to determine Choreographer's detailed performance characteristics for a specific use case, but the eventual decision to use Choreographer needs to be made by an informed customer who can trade off potentially denial of service to some proportion of traffic with the added security benefit.

Sensitivity to latency is one of the issues found, and was highlighted by the Emulytics$^{TM}$ environment used. This sensitivity to latency may be overcome by using more generously provisioned hardware. Further testing would be required to assure Choreographer's successful operation and could help to provide better information for given use cases to prospective users.

The issues with Access Windows Aliasing are a bit more serious, yet can be mitigated by using larger available address blocks. Unfortunately, expanding the address space may in turn create a different performance issues under heavy load, depending on the environment.

Ultimately, scalability is the critical concern when deploying Choreographer. It is sensitive to Denial of Service attacks when receiving a high volume of traffic, in part because libpcap is lossy at high traffic volume. However what constitutes a "high volume of traffic" is largely dependent upon the available resources for the Choreographer server. With a large-enough server, it is possible that only the most high-volume environments could experience Denial of Service conditions. With existing resources, we were able to model approximately 7500 discrete clients, generating on the order of 4 to 5 sessions/minute per client. We can't speak to other issues that may arise in environments with considerably more concurrent clients or discrete connections.

We recommend that any further operational testing or red team testing be conducted to address a specific use case (e.g. server resources, traffic patterns, etc.) to assure Choreographer can perform acceptably without massively over-provisioning the server resources or having a different performance bottleneck emerge.